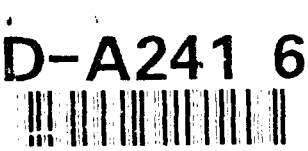


ANNUAL REPORT

VOLUME 4

TASK 4: SOFTWARE DEVELOPMENT

REPORT NO. AR–0142–91–002

September 24, 1991

GUIDANCE, NAVIGATION AND CONTROL

DIGITAL EMULATION TECHNOLOGY LABORATORY

Contract No. DASG60–89–C–0142

Sponsored By

The United States Army Strategic Defense Command

COMPUTER ENGINEERING RESEARCH LABORATORY

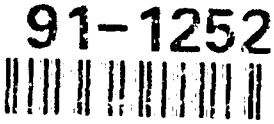
Georgia Institute of Technology

Atlanta, Georgia 30322–0540

Contract Data Requirements List Item A005

Period Covered: FY 91

Type Report: Annual



UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188

1a REPORT SECURITY CLASSIFICATION Unclassified	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT 1)Approved for public release; distribution is unlimited 2) continued on reverse side
2b DECLASSIFICATION/DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S) AR-0142-91-002	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION School of Electrical Eng. Georgia Tech 6b OFFICE SYMBOL (If applicable)	7a NAME OF MONITORING ORGANIZATION U.S. Army Strategic Defense Command
6c ADDRESS (City, State, and ZIP Code) Atlanta, Georgia 30332	7b ADDRESS (City, State, and ZIP Code) P.O. Box 1500 Huntsville, AL 35807-3801
8a NAME OF FUNDING/SPONSORING ORGANIZATION 8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER DASG60-89-C-0142
8c ADDRESS (City, State, and ZIP Code)	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO / PROJECT NO / TASK NO / WORK UNIT ACCESSION NO

11 TITLE (Include Security Classification)
Guidance, Navigation and Control Digital Emulation Technology Laboratory
Volume 4 (Unclassified)

12 PERSONAL AUTHOR(S)
C. O. Alford, Wei Siong Tan, R. Indaheng, J. Lie, M. Alibakhsh

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Annual	FROM 9/28/90 TO 9/27/91	9/27/91	24

16 SUPPLEMENTARY NOTATION

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

1. PFP Software
   - 1.1 Introduction
   - 1.2 Compilers
   - 1.3 Utilities and Support Engironment
   - 1.4 Planned Enhancement
2. PFP Host conversion to SPARC station
   - 2.1 Introduction
   - 2.2 Plan of Attack
   - 2.3 Development Plan
3. Sequencer Upgrade
   - 3.1 Introduction
   - 3.2 Progress
4. Ada Development Strategy
   - 4.1 Introduction
   - 4.2 Compilation Methodology
   - 4.3 PFP Ada Tasking Mechanism
   - 4.4 Ada Examples
   - 4.5 Development Plan
5. Integrated Parallel Programming Framework
   - 5.1 User Interface
   - 5.2 Compilation and Execution
   - 5.3 Operating and Monitoring System

(continued on opposite side)

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT	21 ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	Unclassified
22a NAME OF RESPONSIBLE INDIVIDUAL	22b TELEPHONE (Include Area Code) / 22c OFFICE SYMBOL

DD Form 1473, JUN 86 *Previous editions are obsolete* SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

Abstract (Continued)

TABLE OF CONTENTS

1. PFP Software

1.1. Introduction

Highly parallel architectures offer opportunities for significant improvements in program execution speeds and reliability. However, these opportunities cannot be realized unless effective program development tools are available.

A good Ada programming environment requires an operating system technology for efficient performance execution of parallel programs on different target parallel machines. Here, we have developed a configurable and portable operating system kernel presenting a "library" of primitives used by the programmer. Portability is essential in light of the multi–CPU nature of any PFP machine, which may contain both special–purpose processors, such as FPP and FPX processor boards, and general–purpose processors like the Intel 386/486, and the Intel 860 processor boards.

Henceforth, we have developed C compilers, linkers, loaders, libraries and special–purpose packages for inter–processor communication and coordination on the target parallel machine that will aid in the development of software for our PFP. The C compilers provide efficient run–time system constructs for effective utilization of the underlying capability of hardware resources.

Due to the special–purpose nature of the target hardware, we have been designing and implementing a variety of tools for hardware use, including low–level device drivers, system monitoring and configuration software, etc. In addition, significant efforts have been expended on facilitating system installation, by use of Unix Makefiles. The environment is structured as a collection of tools sharing a common information store, called the abstract information representation.

The current PFP environment and the associated PFP configurable kernal has supported the development of several real–time applications:

a Satellite Attitude Control Simulation
a 3–DOF Missile Simulation
a Multiple Threat/KEW Interceptor Simulation enumerate (EXOSIM)

1.2. Compilers

1.2.1. C

Current FPP/FPX compiler is fully functional and has successfully compiled and run EXOSIM I Boost phase and EXOSIM II Terminal Phase. A new loader developed by Steve Wachtel has been installed and run succesfully. A number of bugs discovered during the development of the EXOSIM simulation have been fixed. The following describes some the bug fixes.

The problem of comparing two operands where the left side operand is an array variable and the right hand side is a math operation. i.e. x[i] > a + b has been fixed. Previously, when the left side of the comparison was an array type with indexing, the compiler first evaluated the right side and did not store the result in the temporary location. When it finished computing the left side index, the compiler assumed the previous result was still in the pipeline which is an incorrect assumption.

The compiler stack handling routine has been fixed to handle a function call with a division operation in the parameter list. i.e. x = func(a/b);

The FPX compiler is now capable of handling true double precision input output.

The FPX compiler produced garbage value when undeflow is detected and zero when overflow is detected. The compiler has been fixed to produce zero when underflow is detected and inf if overflow is detected.

In addition, a MOD function has been built–in to the compiler and it is currently fully operational.

1.2.2. Ada

See section 3 for details

1.3. Utilities and Support Environment

1.3.1.PFP Makefile

A fully customized "make" utility has been written to automate the process of producing executables for all the processors (FPX, FPP, 386, 486, 860). Regardless of the programming language used, the source code is directed to an appropriate translator to produce C code, which is then fed into an appropriate compiler to generate the target object code for execution. Currently, the programming languages supported are Ada, Pascal, C, and Fortran. A crossbar compiler is used to compile the specification of a communication configuration into interconnection patterns between processing elements.

1.3.2.Fortran to C translator

The Fortran to C translator from AT&T has been modified to recognize the PFP interface primitives. It is a fully functional translator which has been incorporated into the front–end PFP software development environment.

1.3.3.Fortran to Ada syntax translator

Originally developed by Steve Wachtel to translate Fortran code to C and modified by Rani Indaheng and Jackson Lie to produce Ada code. The current version uses a direct one to one map-

ping syntax translation scheme to produce Ada code from Fortran. It does not fully support all the advanced Ada features yet. Improvement will be made to support all Ada features in the future.

1.3.4.Device Drivers

The device drivers do not require any new development work at this time. The current drivers will be maintained and upgraded as necessary. They are currently sufficient to run EXOSIM simulation.

1.3.5.Kernel Support

No new development work is expected. Maintenance and upgrade will be provided on an as needed basis.

1.4. Planned Enhancement

1.4.1.PFP Makefile

A fully functional Fortran to Ada syntax translator, and the front–end Ada compiler and the intermediate Ada to C code generator will be ported to Sun–386 and added to the "make" utility.

1.4.2.Next generation C compiler

A next generation FPP/FPX C compiler is being developed based on the GNU C compiler provided by the Free Software Foundation. A new assembler is also being developed for this next generation FPP/FPX C compiler. This new generation C compiler will produce highly optimized code.

2. PFP Host conversion to SPARCstation

2.1. Introduction

The prototype PFP machine used Intel processors and an Intel host which ran a proprietary operating system which was not in common use. As computer workstation technology advanced to include virtual memory, networking, and windowed user interfaces it became desirable to include these features in a PFP host. The Sun 386i was selected since it supported these features and it provided backward compatibility with much of the Intel software. Workstation technology has continued to advance with faster workstations available, obsoleting the older workstations. A faster host improves compilation time, and allows better presentations of the data as the real–time programs are run on the PFP.

Consideration for selecting a new workstation are:

1) speed (cpu, disk, memory),
2) workstation cost,
3) easy of hardware interface to the PFP
4) ease of porting the software from the existing host
5) compatibility with industry standards

The Sun SparcStation series of workstations fit will with the above criteria. The Sun SparcStations family also has several different variations which are all compatible, allowing different cost verses options setups to be selected.

2.2. Plan of Attack

The conversion of the host to a Sun SparcStation can be broken into several steps: converting the hardware interface, writing the necessary UNIX device drivers to work with the new hardware interface, converting the processor compilers, and converting the various PFP support utilities. The SparcStation uses little endian byte ordering, and the Sun 386i uses big endian byte ordering. This is expected to be the major hurdle in porting the software from the Sun 386i to the SparcStation.

2.2.1. Hardware Interface

The Sun 386i was interface the the Multibus I based PFP by purchasing a PC–AT bus to Multibus I interface card from the BIT–3 corporation. This card plugged into the AT bus on the Sun 386i and into the PFP master repeater card cage. The card allows a direct memory map of the Sun's memory space into the the master repeater's address space.

The Sun SparcStation does not have a PC–AT bus. Instead, the SparcStation uses a bus referred to as the S–bus. The S–bus is a 32 bit data, 32 bit address, high speed memory mapped bus suitable for attaching high speed peripherals to the SparcStation. BIT–3 manufactures a S–bus to Multibus I interface, which will be used with the SparcStation port. The BIT–3 S–bus card is very similar to the older PC–AT bus interface from a software perspective, simplifying the software port to the Sun 4 SparcStation.

2.2.2. Device driver conversion

The new host will require device drivers to interface to the PFP. Device drivers are the code that becomes part of the operating system, and handles the low level interface to the BIT–3 S–bus to Multibus card. These device drivers provide the necessary code to transparently handle the handshake messages necessary for the Bit–3 interface, and higher level support to allow each processor to appear a unique UNIX device.

The device driver code can be separated into three layers (see Figure 1 below). The lowest

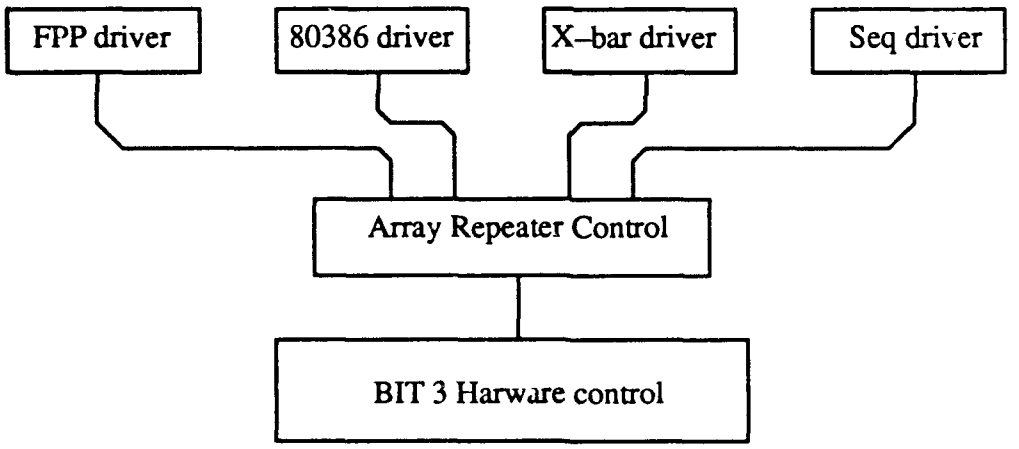


Figure 1. PFP Sun Host Device Drive Hierarchy

layer deals directly with the BIT–3 card and handles the register level interface with the BIT–3 card. The middle layer handles the details of the PFP MultiBus repeater array (reference CERL002–0761–120.1). The highest level layer handles the details of the interface with the specific elements within the PFP, such as FPP processors, FPX processors, 80386 processors, the crossbar(s), and the sequencer(s).

Since the PFP's structure does not change with a conversion to a new host, the types of processors, and the sequencer and crossbar, and the details of the PFP repeater array are unaffected

by switching to a Sun SparcStation host. Thus, the only layer which is effected by the host conversion is the bottom most layer.

Device driver conversion will also be effected by the revision level of the UNIX based Sun Operating System. The SparcStation uses SunOS 4.1.1 while the Sun 386i uses an earlier version, SunOS 4.0.2. These versions have minor differences in how the device drivers are written

2.2.3. Compiler conversion

The compiler source code will be copied directly from the Sun 386i to the Sun SparcStation. The source code for the compilers will then be compiled on the SparcStation, and used to generate a series of test programs. These test programs will also be compiled on the Sun 386i. The object code and binary code resulting from the compilation will then be compared to determine whether any porting problems exists. The majority of the porting problems are expected to arise from the differing byte order between the Sun 386i and the Sun SparcStation.

2.2.4. Utilities conversion

The utilities will be converted using the same strategy as for the compiler conversion.

2.3. Development Plan

To develop the system, a "mini–PFP" will be set up. This mini–PFP will be a one or two processor version with all of the interfaces in place. This mini–PFP will be used to test the initial device drivers and the compilers ports. Final testing on a full PFP will be minimized to prevent impact on the schedules of the various application and simulations that run on the PFP.

The project will begin by purchasing a BIT–3 S–bus interface card and connecting up a test setup on a sun SparcStation. The device driver work can proceed in parallel with the compiler development since the compilers can be run on a SparcStation, and the resulting code can be transferred over a network to a Sun 386 host, eliminating the need for a SparcStation to start the compiler conversion. The actual order of work will depend on the availability of manpower for this effort.

3. Ada Development Strategy

3.1. Introduction

A commercially available validated Ada compiler has been incorporated into our front–end Ada development environment. We licensed both the executable Ada front–end and the intermediate Ada to C generator source code from Irvine Compiler Corp. The current version of the Ada front–end runs on Sun–3. Effort to port the Ada development environment to Sparcstation is under way.

Since we are consistently converting codes written in Fortran to Ada, we are developing a simple Fortran–to–Ada syntax translator that will allow us to automate this repetitive process.

As part of our Ada development environment, we are developing a PFP Ada tasking mechanism that will allow Ada tasks to run in parallel on any native machines with a validated Ada compiler. This mechanism will emulate the PFP inter processor communication sequencer; thus, providing software developers a capability of testing parallel programs on any native machines before porting the same programs with little or no modification to the PFP

3.2. Compilation Methodology

The Ada Compilation Methodology consists of three parts, as shown in Figure 2. The front–end is a validated Ada compiler that reads in Ada code and produces the intermediate representation of the code.

The second part of the Ada Compilation Methodology is the Ada–to–C generator. The intermediate code produced by the front–end is fed into the Ada–to–C generator to produce a C code. The original code generator produces C code that is targeted for machines with eight bit Basic Machine Unit (the smallest addressable unit). Since our PFP BMU size is 32 bits, we modified the code generator source code to produce 32–bit C code.

In order to generate a highly optimized C code, the Ada–to–C generator requires that the configuration of the target machine be defined. We have obtained a copy of the configuration file for Sun–386 from ICC and have modified it to our PFP configuration.

Finally, the last part is the PFP C compilers. They produce executable codes for either GT–FPX, GT–FPP, GT–386, GT–486, or GT–860. The FPP/FPX compiler is developed in house to produce executable code that will run on the single precision FPP processor or the double precision FPX processor. The gcc compiler provided by the Free Software Foundation is used to produce executable code for 386 and 486 processors. Finally, we are in a process of modifying the gcc compiler to produce executable code for the 860 processor.

The effort to compile and translate Ada code to C code reduces not only our maintenance cost by allowing us to maintain only the back–end C compilers, but also reduces the development costs. All of the front–end compilers (including the front–end Fortran compiler) are commercially available and validated compilers. These front–end compilers have been extensively tested in the field by many other vendors.

A Fortran to C translator originally developed by Steve Wachtel has been modified by Rani Indaheng and Jackson Lie to generate Ada code. The current version of our Fortran–to–Ada translator is capable of translating codes in Fortran syntax to Ada. It translates Fortran syntax one–to–one to Ada syntax, and it only supports a subset of Ada rules. It does not take the advantage of most of the advanced Ada features. However, at present time, we are satisfied with the Ada code produced by this translator. Further enhancement will be added in the near future.

3.3. PFP Ada Tasking Mechanism

The PFP Ada tasking mechanism consists of three parts, as shown on Figure 3. The first one is the buffer package. Each task or process is assigned an input and an output buffers. To communicate with other tasks, an Ada task sends a message to its output buffer and it does not need to specify who the receiving task is. When the output buffer is full, the task will wait and poll until the buffer is available, and then writes to it. Once the task finishes writing the message to the output buffer, it can continue its execution. To receive a message from another task, it reads its input buffer without any knowledge of who the sender is. In the case where the task is ready to receive a message and the input buffer is empty, it will wait and poll until the buffer is not empty and then reads the message from the buffer. Once it finishes reading the message, it is free to resume execution.

For each project, a user must write an inter–process communication (crossbar) map that will provide the communication pattern between tasks. In other words, the crossbar file will tell the Ada sequencer emulator which tasks are sending messages to other tasks on a certain CPU cycle. The inter–process communication file is a simple script–file that does not require a user to learn any new language or any new methodology. The file consists of two parts. The first one tells the sequencer emulator which task is assigned to which processor (virtual processor). The second part provides the information of who the sender is and who the receiver is. Below is an example of a crossbar file.

```
process_1 is task_1.fpx on x0
process_2 is task_2.fpp on x1
process_3 is task_3.386 on x2
process_4 is task_4.486 on y0
loop
cycle process_1 —> process_2, process_3;
```

```
cycle process_2 —> process_3;
     process_1 —> process_4;
```

The second part is the scanner package. This package will read and compile the specification of a communication configuration defined in crossbar file into interconnection patterns between processing elements.

Finally, the inter–process communication package will emulate the PFP inter–process communication sequencer and does the message transfer from the output buffer of the sending task to the input buffer(s) of the receiving task(s) according to the communication configuration defined in the crossbar file.

3.4. Ada Examples

We have developed an Ada parallel version of the EXOSIM boost phase to test the Ada development environment. The steps that are involved are as follows.

We took the uni–processor version of the EXOSIM written in Fortran and broke it down to twenty four separate modules. Each module was translated to C by the Fortran–to–C translator from AT&T. We then took the C codes and compiled them using our FPP/FPX compiler. The executable codes were then tested module by module. Finally, we integrated all twenty four modules and run them on PFP in parallel and in real–time.

Once the twenty four Fortran modules were running successfully on PFP, we translated the Fortran codes to Ada, manually, and run them through the front–end Ada compiler and Ada–to–C code generator. The C codes were then compiled using our FPP/FPX compiler and run them on PFP in parallel and in real–time.

At present time, the size of the Ada version of the EXOSIM object codes is about fifty percents larger, on the average, than the Fortran object codes. The speed of the Ada version of the EXOSIM, however, is about thirty two percents faster than the Fortran version. Two main factors constitute to this improvement. One is that the Ada programming language has more advanced features that will allow one to write a more efficient code. Second, the intermediate Ada–to–C code generator does a better job of producing better C codes.

3.5. Development Plan

Ada development plan is shown on Figure 4.

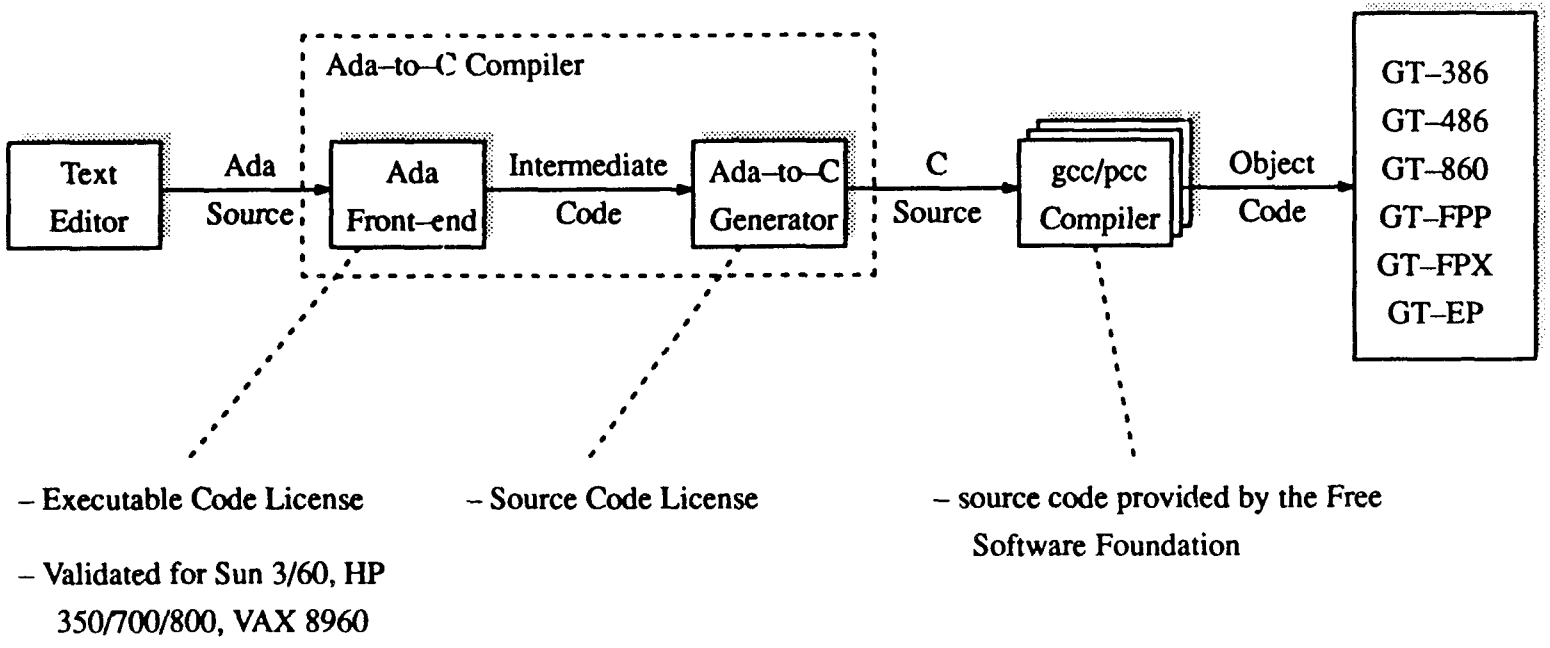


Figure 2. Ada Compilation Methodology

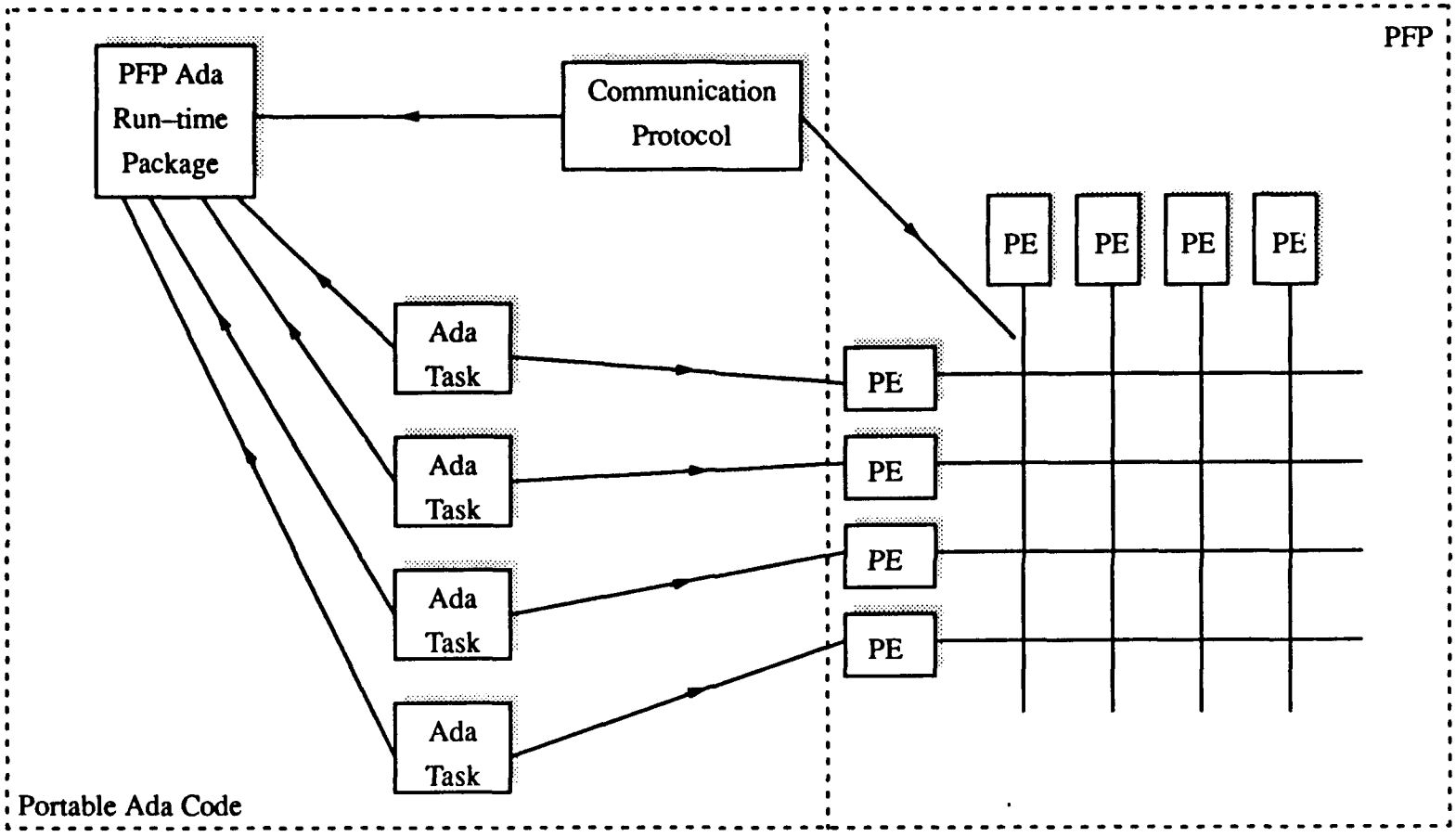


Figure 3.PFP Ada Tasking Mechanism

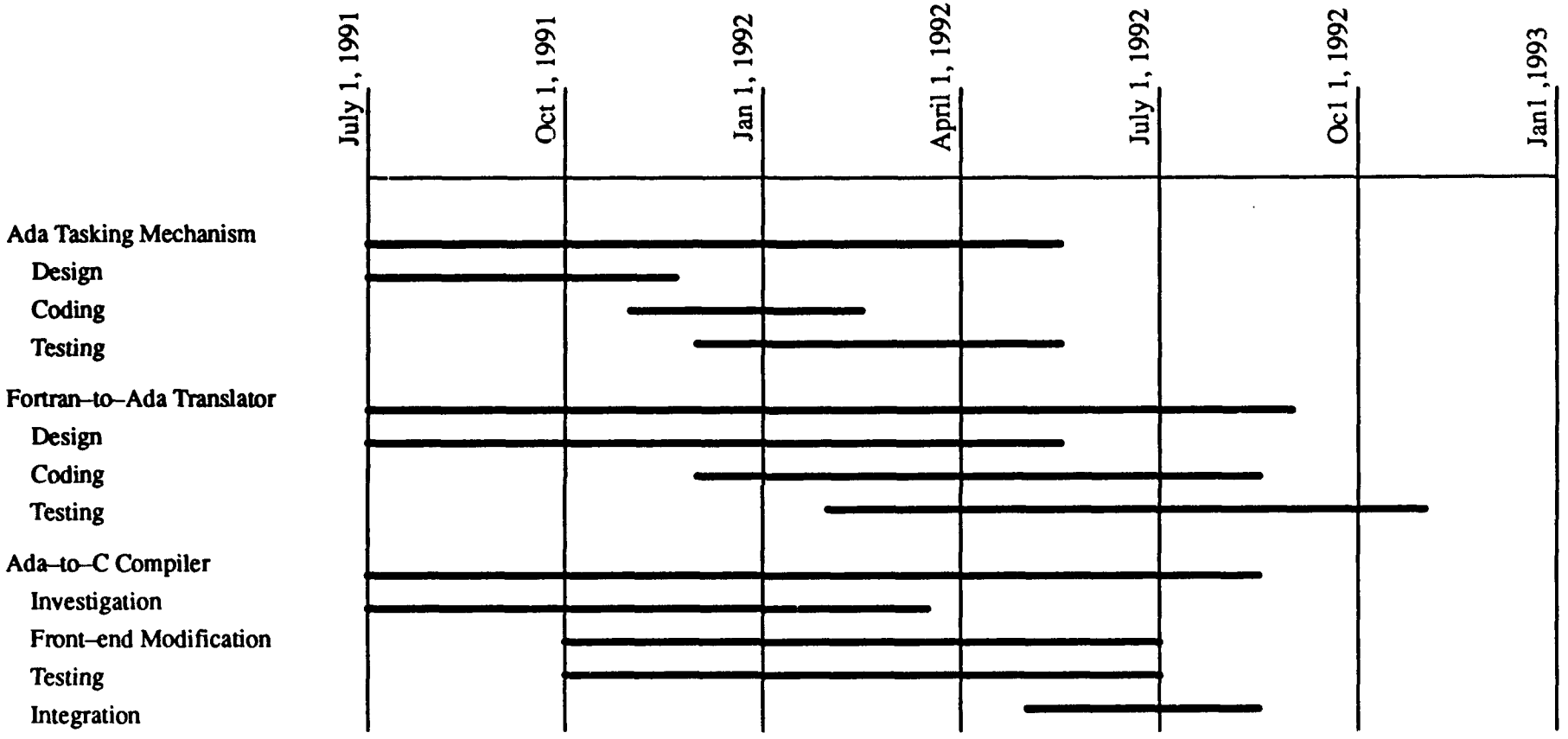


Figure 4. PFP Ada Development Schedule

4. Integrated Parallel Programming Framework

Georgia Tech is devising an integrated parallel programming framework (IPPF) for the development of software for the special purpose parallel processor architectures. The IPPF serves as an entity for the integration of diverse hardware components and provides a consistent interface for effective exploitation of hardware capabilities. The IPPF consists of four components: user interface, compilation and execution, operating and monitoring system, and database and tool integration. The overall software architecture of the IPPF is shown in Figure 5.

4.1. User Interface

The conventional programming method uses direct text editing of program source code. This mode of programming is still supported in the IPPF environment. However, the primary method of programming is through a block editor and block diagram editor. Using the block editor, the user creates and defines basic functional programming blocks. Each functional block is represented by a graphical block diagram. Data flowing into and out of the block are represented by input and output connection ports. The behavior of the block is represented by a self -contained code segment with receive commands for information flowing into the block and send commands for information flowing out of the block. The block diagram editor allows a user to assemble predefined functional blocks and connect the blocks into a higher level system of functional blocks. Using this method, a complex application program can be graphically and hierarchically developed.

The configuration editor and monitoring specification allows a user to effectively control hardware resources and specify specific monitoring information for an application run. The default hardware configuration is automatically extracted from the application block diagram. Manual configuration is only needed if optimization around a particular hardware configuration is desired.

A commercial package, namely Matrix–X, is used as the front end graphical block diagram editor. An additional module Autocode is used to generate C and Ada source code for the designed blocks. These packages will run on a SUN host running SUN's version of Unix operating system.

Matrix–X is a powerful, programmable, matrix calculator with a graphical interface. It is capable of solving complex, large scale matrix problems in any engineering discipline. In addition to matrix analysis functions, it provides a rich set of design and analysis functions for classical input/output control and modern state–space control problems.

4.2. Compilation and Execution

Regardless of the form of user interface, either through block diagram editing or direct program editing, the eventual output from the user interface is program source code. The source code

is directed to an appropriate compiler to generate the target object code for execution. The programming languages supported are Ada, Pascal, C, and FORTRAN. A crossbar compiler is used to compile the specification of a communication configuration into interconnection patterns between processing elements. Each processor type requires a separate compiler, linker, and loader. Processor specific low level utilities are incorporated into the environment database.

4.3. Operating and Monitoring System

The operating system provides efficient run–time system constructs for effective utilization of the underlying capability of hardware resources. It provides a concise interface between the various programming languages (Ada, C, Assembly language, etc.) and the underlying architecture of each type of processor element. It also provides a facility for exception handling and error recovery mechanisms. The operating system gives each processor element the ability to execute multiple tasks. This is especially important when the number of functional blocks exceeds the number of available processor elements.

The monitoring system provides a useful mechanism to obtain feedback from an application run. It provides capability for a systematic display of application results and a structured facility for real–time data collection. The monitoring system also serves as the interface for system debugging.

4.4. Database and Tool Integration

The software database provides a repository for the block diagrams, configuration information, monitoring specification, language–specific package library, target–machine specific utilities, operating system constructs, and monitoring systems. The database serves as an integration tool for the other three components of the IPPF. A database interface provides constructs to create, access, update, and display the information in the database.

4.5. Development Strategy

4.5.1. Matrix–X Integration Plans

Matrix–X is used as a graphical front end to design the systems. The Autocode is used to generate the C or Ada source code based on the block diagram designs. These source modules are then processed and the interface and interblock communication information is then updated to a database. The information in the database is subsequently used to generate the necessary crossbar code. The initial source modules are also modified in the proper manner to include the interprocessor communication calls. The programs are then compiled, loaded and executed in parallel on the PFP.

4.5.2. Development Schedule

The development and integration of Matrix–X will be done in two phases. The first phase will be completed for the C language sources, and the second phase will include the Ada version. Initially Matrix–X will be moved to a SUN platform from the VAX stations. This is expected to be completed approximately by the end of August 1991. The designing and development of the crossbar code generator and the C source analyzer will immediately begin following the successful installation of the Matrix–X software on the SUN platform. The work on the Ada version will commence following the successful development, testing and debugging of the initial C version.

4.6. Technologies for Parallel Programming Environments

The successful development of a PPE requires the following technologies:

> Language and compiler technology for the support of multiple models of parallel programming (when explicitly describing parallelism) and for the automatic or semi–automatic parallelization of programs. Here, we are using and developing Ada and C compilers, linkers, and loaders, enhanced with libraries and special–purpose packages for inter–processor communication and coordination on the target parallel machine.

> For program generation, analysis, and improvement: programming environment, database, and visualization technology for the representation, sharing, and display of information about the parallel program, its execution environment. and its run–time performance. This is the PPE being developed in this research.

> Performance analysis, program specification techniques, and, perhaps, Artificial Intelligence technology (1) for performance modeling of the parallel program, (2) for expression of relevant performance attributes of parallel programs or of program invariants runtime not to be changed during performance tuning, and (3) for relating models and measurements to actual program code as well as for suggesting and making changes to such code. Here, we will offer simple means of performance evaluation and program visualization, coupled to a graphical interface used for program development.

> Operating system technology for efficient performance monitoring and for the efficient execution of parallel programs on different target parallel machines. Here, we have developed a configurable and portable operating system kernel presenting a "library" of primitives used by the programmer.

Portability is essential in light of the multi–CPU nature of any PFP machine, which may contain both special–purpose processors (such as the FPP) and general–purpose processors like the Intel 386 or 860 boards.

4.7. Summary

4.7.1. Parallelism and application domains.

Highly parallel architectures offer opportunities for significant improvements in program execution speeds and reliability. However, these opportunities cannot be realized unless effective program development tools are available. A parallel programming environment (PPE) differs from conventional program development systems in several ways.

For the explicit expression of parallelism, the programming model presented to the programmer should address the specific properties of the programmer's application domain. For example, in real–time simulations, low–level control functions are easily described as statically decomposed collections of communicating functional blocks. This suggests that a useful programming model is one that presents the functional (possibly replicated) building blocks in the application using graphical descriptions. This is the approach we will pursue for the PPE being constructed for the PFP, in conjunction with visual illustrations of the performance effects of such decompositions.

More importantly, one of the environment's attributes will be its ability to exploit application domain–specific knowledge for assistance in parallel programming. For example, when performing resource allocation for PFP's real–time simulations (e.g., mapping functional blocks to processors), the system will use built–in mapping functions, thereby removing from programmers the responsibility of computing and enforcing such mappings.

4.7.2. Performance evaluation and improvement.

Since the primary objective of parallel computing is performance improvement, a PPE must assist the programmer in gaining understanding of program performance on the target parallel machine. This implies (1) that tools for program monitoring, performance evaluation or prediction and for the visualization of performance information should be integral parts of the programming system and (2) that programmers should be assisted in making changes to their parallel applications in response to such evaluations or predictions — termed program tuning. Specifically, we will assume that the PPE should provide a general framework that makes effective use of a wide variety of performance display, evaluation, and visualization tools.

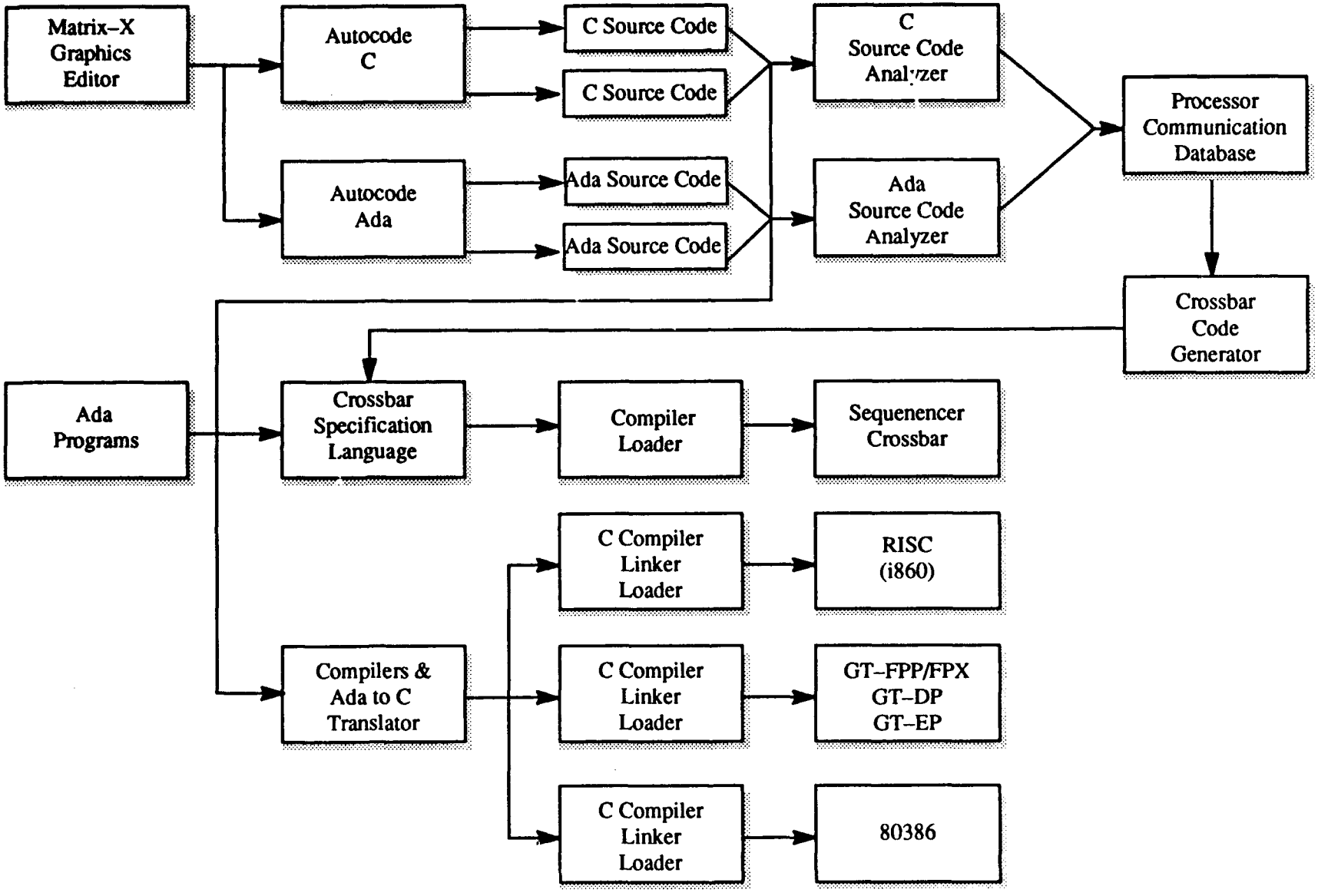


Figure 5. Integrated Parallel Programming Framework

5. GT–EP Software

The GT–EP software development flowchart is shown in Figure 6. Each component is described in the following sections.

5.1. Pascal Compiler

The GT–EP Pascal Compiler was developed to generate object codes for the Genesil multichip simulation of the GT–EP processor. To minimize the development time, only a subset of standard Pascal is implemented. This document provides a brief description of the constructs supported by the compiler, the files that make up the compiler, and the command line to invoke the compiler.

5.1.1. Compiler Features

5.1.1.1. Data Types

The compiler supports three basic data types: real, integer, and boolean. Real types are represented by 32–bit IEEE single precision floating point numbers and integer types are represented by 25–bit signed magnitude fixed point numbers. Boolean false is represented by 0 and true by non–zero. The compiler accepts arrays of reals and arrays of integers. No user–defined datatypes are allowed. The pascal "type ..." construct is not supported.

5.1.1.2. Expressions

Arbitrary arithmetic and boolean expressions are supported. The supported operators for these expressions are described in the following sections.

5.1.1.2.1. Arithmetic Operators

For real data types, the operators supported are +, –, *, and /. For integer data types, the operators supported are +, –, *, or, and, shl, shr, rol, ror, and xor. Conversion operations supported are trunc and round.

5.1.1.2.2. Boolean Operators

Boolean operators supported for both real and integer data types are >, <, >=, <=, and =. Boolean operators for boolean data types are "and", and "or".

5.1.1.2.3. Math functions

The math functions supported are sin, cos, asin, acos, tan, atan, ln, exp, and sqrt.

5.1.1.2.4. Assembly Instructions

All the assembly instructions in the GT–VIAG and GT–VDAG programming model documents are generally supported by the compiler. The load instructions are implemented as procedure calls and the store instructions are implemented as function calls.

5.1.2. Compiler Makeup

The following are the compiler source files:

arith.pas,
check.pas\
code_gen.pas\
compile.pas\
declare.pas\
exprsion.pas\
exprtree.pas\
fetch_tk.pas\
for_stat.pas\
global.pas\
hex_conv.pas\
if_while.pas\
init.pas\
io.pas\
lib.pas\
mainbody.pas\
procedur.pas\
std_proc.pas\
stdprocd.pas\
symbol_t.pas\
utility.pas
bfilter.pas

The top level file is compile.pas and bfilter.pas. Bfilter.pas is a standalone program. All the other source files are used by compile.pas. The file global.pas contains all the definitions of the global variables. Compile.pas receives a .pas file and produces a .fpp file.

The execution of bfilter.pas collapses all forward branch references in the .fpp file and produces a .ep file.

5.1.3. Command Line

To execute the compiler invoke

compiler <filename> [d]

The compiler automatically appends .pas to the filename and expects an input file with that extension. The optional d flag is used to direct the compiler to print out the tree structure of the compiled code.

The compiler produces a xxx.err file and a xxx.fpp file . If a compilation error is detected, the compiler directs the user to the xxx.err file for a listing of the error message. The compiler scans the xxx.pas input file and writes to the xxx.err file one statement at a time. If a syntax error is detected, the compiler writes an error message to the xxx.err file and aborts the compilation process. If the xxx.pas contains no syntax errors, the compiler simply generates a copy of the xxx.pas file in the xxx.err file.

The object code is listed in the .fpp file written as a standard ASCII file. The format of the file is listed in Appendix A.

To resolve forward branch reference, the following command line is used

bfilter <filename>

The program expects a file with .fpp extension and produces a file with .ep extension

A user document for the Pascal Compiler was generated and sent to Harris along with the compiler executable and source files. Both the PC–DOS and the Sun–Unix versions of the source files were included.

The compiler that runs on the PC has been successfully ported to run on the SparcStation. Some careful considerations were taken to ensure that only one version of the source files needs to be maintained A file driven stream editor and a makefile are used to automatically convert the Pascal source files to C source files and compile an executable image to run on the SparcStation. Changes can now be made on the original source files and a Sun version can be produced without the programmer's manual intervention. The same mechanism can be used to port the compiler to the other Sun architectures.

The following functions were added to the Pascal Compiler: "xor", "and", "or", "shl", "shr", "rol", "ror", and "not". The functions that need to be added is "trunc".

5.2. C Compiler

Basis Technology Corporation in Boston, Massachusett had signed a subcontract agreement with Georgia Tech to develop a C compiler. The compiler will comply with the ANSI C standard. The C compiler will be based on the "gcc" compiler front–end provided for by the Free Software Foundation. Both the ANSI C and the GNU standard test suites will be used to verify the compiler. The expected completion date for the compiler is February 1992.

5.3. Ada Compiler

Georgia Tech will jointly develop a 'validated Ada compiler' with Irvine Compiler Corporation for the GT–EP processor. Work on the compiler is expected to start in Febuary, 1992 with a completion date of January 1993.

The Ada compiler produces an intermediate C code which is compiled using a C compiler to the target GT–EP object code. This approach significantly reduces the cost of providing a fully validated compiler for the GT–EP processor. The Ervine Ada compiler currently support a wide variety of target processors including the Sun and HP workstations, and the Intel i960 embedded processors.

5.4. Object Code

The unix a.out standard binary file format will be used for the GT–EP object code. Currently the Pascal compiler produces an ASCII file each line expressing the values of the 12 GT–EP opcode fields. The compiler eventually will be modified to produce object code in the a.out binary file format. The C compiler will also produce the object code in the a.out format.

5.5. Linker/Loader

The Linker/Loader combines multiple object files, resolves external references, produces an execution file, and downloads programs to the GT–EP for execution.

5.6. Library

The library will contain standard math functions such as sin, cos, and other math functions that are not directly supported by the GT–EP instruction set. It will eventually expand to include optimized function such as those required to solve differential equations or to compute FFT efficiently.

5.7. Run–time Kernel

The run–time kernel will include interrupt service routines, task scheduler, and communication primitives. It will also include functions to perform system tests and perform services for user programs which require kernel priviledge. The run–time kernel will be responsible for servicing and scheduling real–time I/O functions.

5.8. Host Utilities

The Host Utilities will include basic routines for the GT–EP processor to communicate with the host.

5.9. Interactive Programs

Interactive programs serve as a gateway between the users and the GT–EP processor. They run on the host and communicate with the GT–EP processor using the basic host utilities.

5.10. Sun Host

The host processor for the GT–EP processor will be a SPARC based Sun workstation.

5.11. Hardware Diagnostic

The hardware diagnostic software checks all the functional modules of the GT–EP and report the status of each module. This is an absolutely necessary first step to verify that hardware is functioning properly before any application software is run.

5.12. Software Debugger

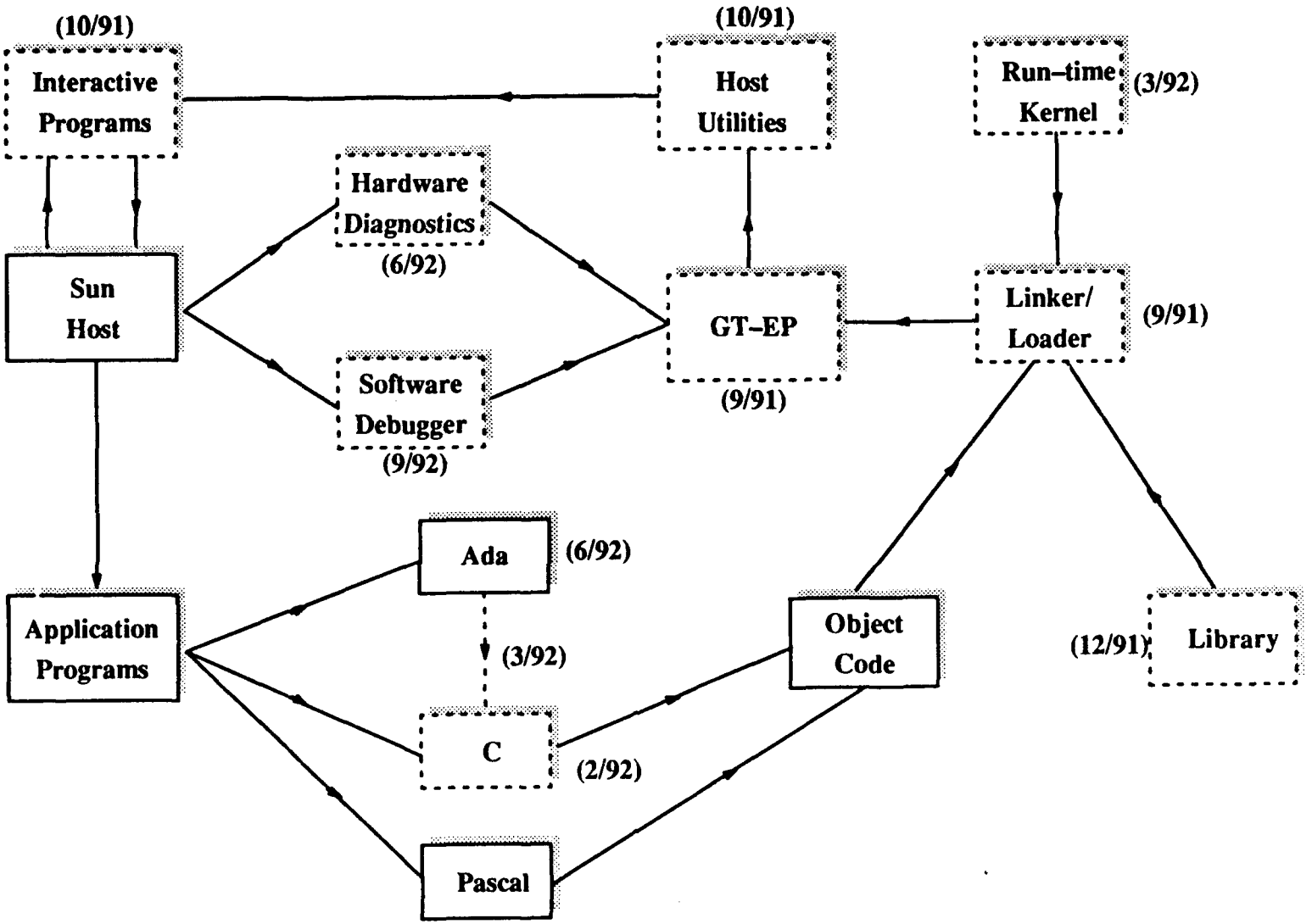
As the complexity of the application programs increases, it becomes necessary to include a software debugger to speed up debugging process. Basis Technology Corporation will follow the C compiler development with the development of a software debugger for the GT–EP processor. The development effort is expected to take six to nine months after the completion of the C compiler.

5.13. Development Status

The Pascal Compiler was used to compile programs for the GT–EP evaluation test board. Three small test programs had been successfully run on the GT–EP processor. Work is in progress to integrate the GT–EP evaluation test board with the Sun host.

The contract with the Basis Technology Corporation to develop with the C compiler has been signed. The development effort is now in progress.

The EXOSIM boost phase had been converted Ada to run on the Parallel Function Processor. The Ada methodology for the Parallel Function Processor is the same as that for the GT–EP processor. It is expected that the GT–EP development effort will greatly benefit from the Ada development which had already under way for the Parallel Function Processor. Contract negotiation with Ervine to provide a fully validated Ada compiler for the GT–EP processor is in p ogress.



Note: Target dates for completion are given beside each block.

Figure 6. GT–EP Ada/C Software Development Flow Chart